Chapter 5: Python Fun with Functions!

Contents

Time required: 90 minutes

DRY

Don't **R**epeat **Y**ourself (**DRY**) is a principle of software engineering aimed at reducing repetition of software patterns. It you are repeating any code, there is probably a better solution.

Visualize and Debug Programs

The website www.pythontutor.com helps you create visualizations for the code in all the listings and step through those programs one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. This will help you to better understand the behavior of the programs we are working on.

This is a great way to debug your code. You can see the variables change as you step through the program.

www.pythontutor.com

Learning Outcomes

Students will be able to:

- Define the components of a function header
- Define and produce a function body
- Understand and use scope and lifetime
- Understand argument passing and use
- Understand and properly call methods, void and value returning
- Understand and use the proper return syntax
- Understand how to use returned values in your calling code
- Understand how to create and use modules

Comments and Docstrings

A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. Whenever string literals are present just after the definition of a function, module, class, or method, they are associated with the object as their `__doc__` attribute. We can later use this attribute to retrieve this docstring. This is a common Python code documentation process.

Docstrings are also a nice-looking way to comment your code.

Single Line Docstrings

```
''' Three single quotes is pretty fun. '''
""" We can also use three double quotes. """
```

A docstring can use ''' or """ to begin and end a docstring.

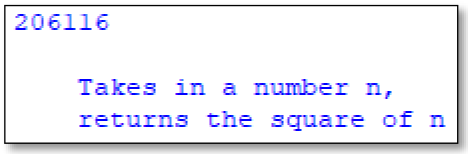
Multiple Line Docstrings

```
'''
Takes in a number n, returns the square of n
Second line
'''
```

Print Docstrings

```
def main():
    number = 454
    number_squared = square(number)
    print(f'{number} squared is {number_squared}')
    # Print the doc string
    print(square.__doc__)
def square(n):
    '''
        Takes in a number n,
        returns the square of n
    '''
    return n**2
main()
```

Example run:



```
206116

    Takes in a number n,
    returns the square of n
```

Docstrings can also be used for program headings and pseudocode.

Why Functions?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

- **Clearer Code:** Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read, understand, and debug.
- **Simpler Code:** Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- **Easier Debugging:** Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.

- **Code Reuse:** Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Elements of a Function

A function is a named block of reusable code that performs a single, related job. Functions provide better modularity for your application and a high degree of code reusability. As you already know, Python gives you many built-in functions like `print()`, etc. but you can also create your own functions. These functions are called user-defined functions.

To get a function to do its job, you “call” it, with some appropriate parameters if the function requires them. The idea is that you do not need to have knowledge about how a function performs its action. You only need to know three things:

- The name of the function
- The parameters it needs (if any)
- The return value of the function (if any)

Syntax for a Python function definition.

```
def name( parameters ):
    """ optional docstring """
    codeBlock
    optional return [expression]
```

Here is a program that demonstrates the code flow with a user created function being called. Be sure to visualize this code on your own step by step to help your understanding.

```
def simple_function(hello):
    '''
    Prints Hello World!
    '''
    print("Hello World!")
# End function definition

# Call the function
simple_function()
```

Function Name

Each function has a name. Like a variable name, a function name may consist of letters, digits, and underscores, and cannot start with a digit. Almost all standard Python functions consist only of lower-case letters. A function should expresses concisely what the function does.

When referring to a function, it is a convention to use the name, and put an opening and closing parenthesis after the name. Functions are always called in code with parentheses.

Void and Value-Returning Functions

Void function: does a specific job and then terminates.

Value returning function: returns a value back to the point where it was called.

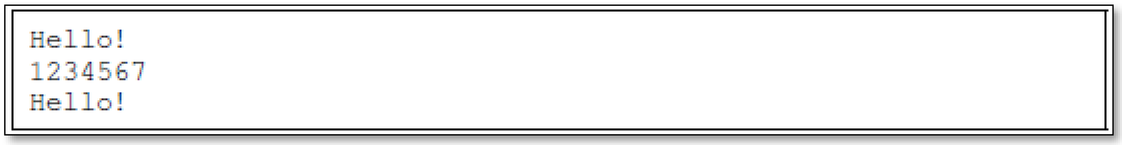
Creating Functions

Functions are defined with the `def` statement. This is called the function header. The function header ends with a colon.

The code after the header is called a block. The block of code is indented below the `def` statement. The indentation is important, that tells Python where the function begins and ends.

Here is a simple function that prints Hello.

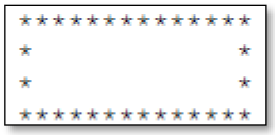
```python
def print_hello():
    print("Hello!")

print_hello()
print("1234567")
print_hello()
```



```
Hello!
1234567
Hello!
```

The first two lines define the function. In the last three lines we call the function twice.

One use for functions is if you are using the same code over and over again in various parts of your program, you can make your program shorter and easier to understand by putting the code in a function. For instance, want to print a box of stars like the one below at several points in your program.



```
***************
*             *
*             *
***************
```

Put the code into a function. Whenever you need a box, just call the function rather than typing several lines of redundant code. Here is the function.

```
def draw_square():
    print("*" * 15)
    print("*"," "*11, "*")
    print("*"," "*11, "*")
    print("*" * 15)
```

One benefit of this is that if you decide to change the size of the box, you just have to modify the code in the function. If you had copied and pasted the box-drawing code everywhere you needed it, you would have to change all of them.

Python Program Template with Functions

Instead of typing everything from scratch, you can use this Python Program Template to speed up your development process. Copy and paste it into a new Python program file, save it as the name you wish.

```
"""
    Name: program_template_for_functions
    Author:
    Created:
    Purpose:
"""

def main():
    """
        Main program function starts here
    """

"""
    Other function definitions here
"""



"""
    If a standalone program, call the main function
    Else, use as a module
"
if __name__ == "s__main__":
    main()
```

Tutorial 5.1 – A Void Function

Time to create our first function. This program has two functions, `message()` and `main()`. The `main()` function is where the program starts. Most of our programs from this point on will have a `main()` function.

Put in your own favorite saying in this tutorial.

```python
"""
    Name: message_function.py
    Author:
    Created:
    Purpose: Demonstrate a Simple Function
"""

def main():
    """Main program function starts here """

    # Print a blank line
    print("\nThe first message function call.\n")

    # Call the message function
    message()
    print("The second message function call.\n")

    # Call the message function again
    message()
    print("Done")

def message():
    """ Define the message function """

    print("It is not necessary to change. Survival is not mandatory.")
    print("W. Edwards Deming\n")
    print("Press Enter to continue")

    # Pauses the execution until the Enter key is pressed
    input()

# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

Example run:

```
The first message function call.

It's not necessary to change. Survival is not mandatory.
W. Edwards Deming

Press Enter to continue

The second message function call.

It's not necessary to change. Survival is not mandatory.
W. Edwards Deming

Press Enter to continue

Done
```

Scope and Lifetime

Scope refers to visibility. When discussing the scope of a variable, it refers to the places in a program where a variable is visible and can be changed. Lifetime refers to how long a variable exists in memory. Lifetime is closely related to scope.

In general, the scope of a variable is at least the code block in which it is created and all the code blocks that are nested within that code block at a deeper indent level.

Local Variables

What happens in the function, stays in the function.

Let's say we have two functions like the ones below that each use a variable `i`:

```
def function1():
    for i in range(10):
        print(i)

def function2():
    i = 100
    function1()
    print(i)
```

A problem that could arise here is that when we call `function1`, we might mess up the value of `i` in `function2`. In a large program it would be a nightmare trying to make sure that we don't repeat variable names in different functions.

Fortunately, we don't have to worry about this. When a variable is defined inside a function, it is local to that function, which means it essentially does not exist outside that function. This way each function can define its own variables and not have to worry about if those variable names are used in other functions.

Global Variables

Sometimes you do want the same variable to be available to multiple functions. Such a variable is called a global variable. You must be careful using global variables, especially in larger programs. A few global variables used judiciously are fine in smaller programs. Here is a short example:

```python
time_left = 30

def reset():
    global time_left
    time_left = 0

def print_time():
    print(time_left)
```

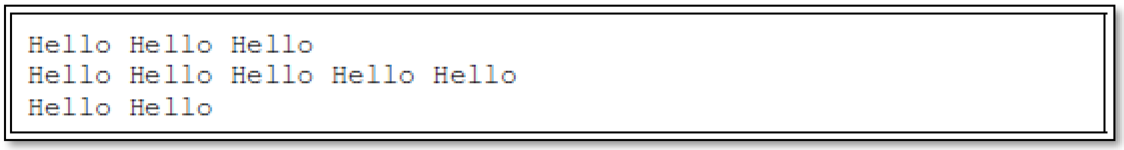
In this program we have a variable `time_left` that we would like multiple functions to have access to. If a function wants to change the value of that variable, we need to tell the function that `time_left` is a global variable. We use a global statement in the function to do this. On the other hand, if we just want to use the value of the global variable, we do not need a global statement.

Passing Arguments to Functions

When we pass values (argument) to functions, we use a local variable called a parameter. Quite often you will hear argument and parameter used interchangeably. The argument sends, the parameter receives. The most important thing is to remember how they work.

Here is an example of a single argument:

```python
# n is the parameter
def print_hello(n):
    print("Hello" * n)
    print()

# 3 is the argument
print_hello(3)
# 5 is the argument
print_hello(5)
times = 2
# times is the argument
print_hello(times)
```

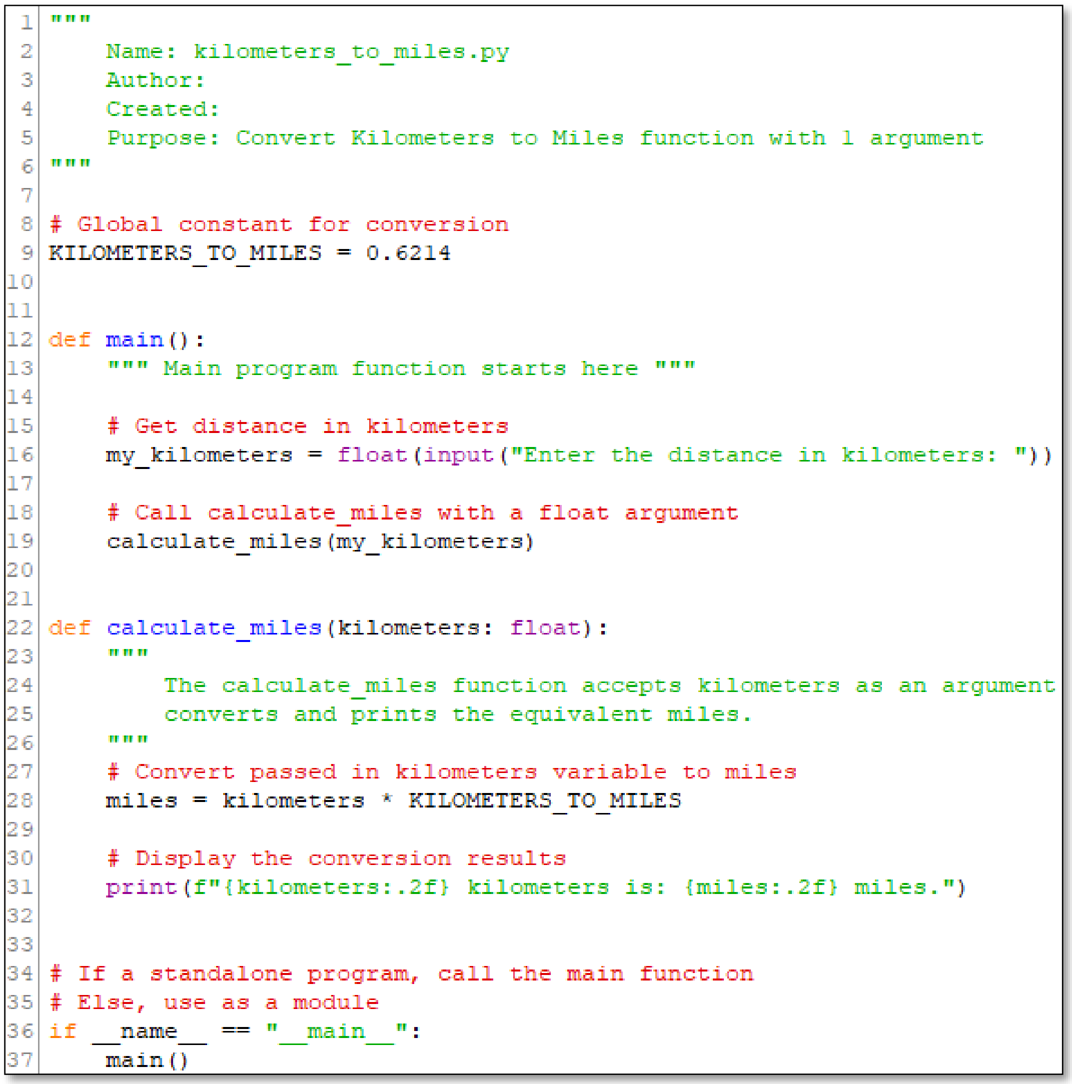


```
Hello Hello Hello
Hello Hello Hello Hello Hello
Hello Hello
```

The first time we call the function `print_hello` with the parameter `n`, we pass 3 to the function and assign it to the local variable `n`. n contains the value 3. The function prints Hello 3x's.

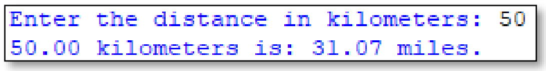
Tutorial 5.2 – Function with a Single Argument

Create and test the following Python program that demonstrates functions with arguments. The `calculate_miles()` function has only one argument. A function can have multiple arguments.



```python
"""
    Name: kilometers_to_miles.py
    Author:
    Created:
    Purpose: Convert Kilometers to Miles function with 1 argument
"""

# Global constant for conversion
KILOMETERS_TO_MILES = 0.6214


def main():
    """ Main program function starts here """

    # Get distance in kilometers
    my_kilometers = float(input("Enter the distance in kilometers: "))

    # Call calculate_miles with a float argument
    calculate_miles(my_kilometers)


def calculate_miles(kilometers: float):
    """
        The calculate_miles function accepts kilometers as an argument
        converts and prints the equivalent miles.
    """
    # Convert passed in kilometers variable to miles
    miles = kilometers * KILOMETERS_TO_MILES

    # Display the conversion results
    print(f"{kilometers:.2f} kilometers is: {miles:.2f} miles.")


# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

Example run:

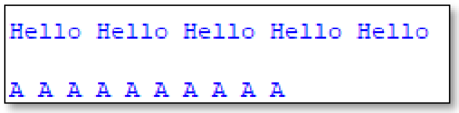


```
Enter the distance in kilometers: 50
50.00 kilometers is: 31.07 miles.
```

Functions can have multiple arguments. The multiple_print function has two parameters: a string and a value. Each are stored in local variables. We then refer to those variables in our function’s code.

```
def multiple_print( string, n ):
    print()
    print( string * n )

multiple_print("Hello ", 5 )
multiple_print("A ", 10 )
```

Example run:



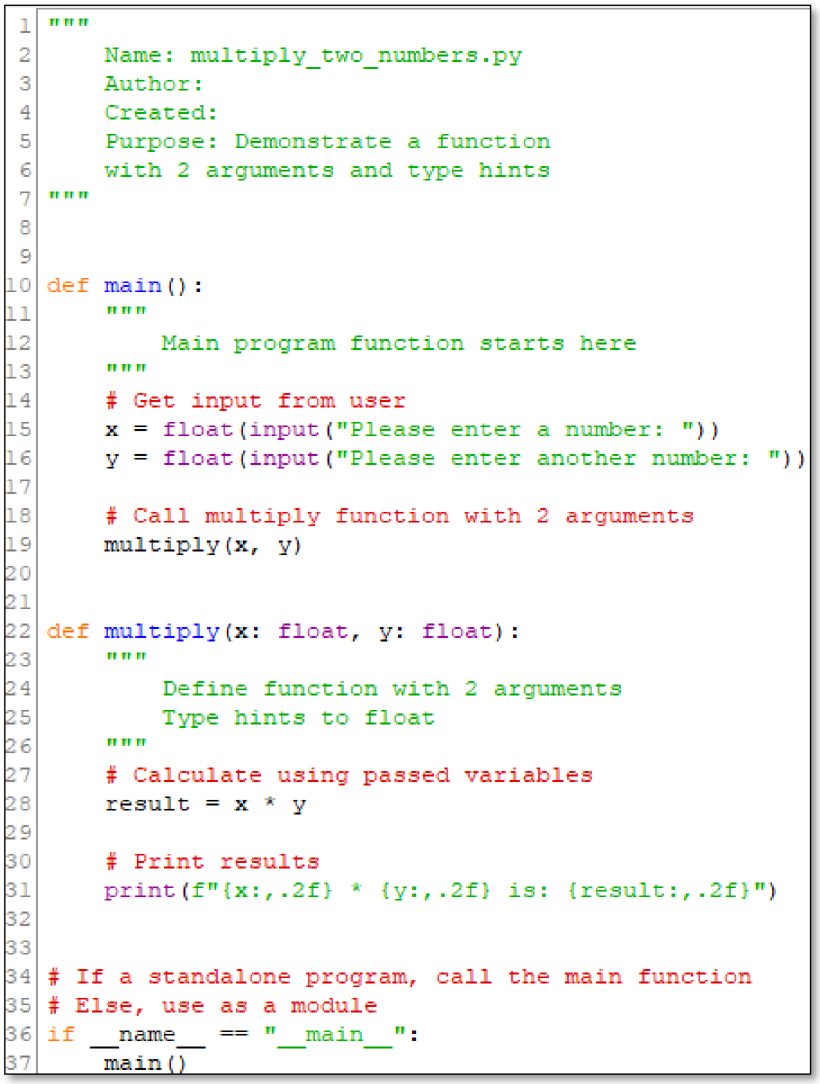
```
Hello Hello Hello Hello Hello

A A A A A A A A A A
```

The following program defines a function that returns the nth root of a number. The first parameter specifies the number for which the root is to be obtained. The second parameter specifies which root is to be obtained.

```
"""
    Name: get_root.py
    Author:
    Created:
    Purpose: Print program title function or module
"""

def main():
    """Call the getRoot function several times"""

    print("square root of 2")
    print(get_root( 2, 2 ))

    print("cube root of 27")
    print(get_root( 27, 3 ))

    print("eighth root of 256")
    print(get_root( 256, 8 ))

    print("sixteenth root of 65536")
    print(get_root( 65536, 16 ))

def get_root( number, root ):
    """
    Returns the nth root of a number
    """
    the_root = number ** ( 1 / root )
    return the_root
# End function definition

# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

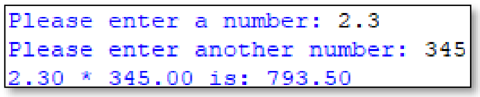
Tutorial 5.3 – Function with Multiple Arguments

The following program demonstrates a function with 2 arguments.



```python
"""
    Name: multiply_two_numbers.py
    Author:
    Created:
    Purpose: Demonstrate a function
    with 2 arguments and type hints
"""


def main():
    """
        Main program function starts here
    """
    # Get input from user
    x = float(input("Please enter a number: "))
    y = float(input("Please enter another number: "))

    # Call multiply function with 2 arguments
    multiply(x, y)


def multiply(x: float, y: float):
    """
        Define function with 2 arguments
        Type hints to float
    """
    # Calculate using passed variables
    result = x * y

    # Print results
    print(f"{x:,.2f} * {y:,.2f} is: {result:,.2f}")


# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

Example run:



```
Please enter a number: 2.3
Please enter another number: 345
2.30 * 345.00 is: 793.50
```

Value Returning Functions

A value returning function performs a job and returns a result.

When you use the command return in a function, that ends the processing of the function, and Python will continue with the code that comes after the call to the function. You can put one or more values or variables after the return statement. These values are communicated to the program outside the function. If you want to use them outside the function, you can put them into a variable when you assign the call to the function to that variable.

Here is a simple function that converts temperatures from Celsius to Fahrenheit.

```
def convert( t ):
    return t * 9 / 5 + 32
print(convert( 20 ))
```

The return statement is used to send the result of a function's calculations back to the caller.

Notice that the function itself does not do any printing. The printing is done outside of the function. That way, we can do math with the result if we wish, as shown below.

```
print(convert( 20 ) + 5)
```

If we had just printed the result in the function instead of returning it, the result would have been printed to the screen and forgotten about, and we would not be able to do anything with it.

Tutorial 5.4 – Functions with Multiple Arguments and Value Returns

Functions can contain any combination and number of arguments and value returns. Create and name this program: **purchase_price_with_functions.py**

```python
"""
    Name: purchase_price_with_functions.py
    Author:
    Created:
    Purpose: Calculate total sale with functions
"""


def main():
    """   Return functions without arguments   """
    purchase_price, quantity = get_input()

    # Return function with arguments
    total_sale = calculate_total_sale(purchase_price, quantity)

    # Void function
    display_sale(purchase_price, quantity, total_sale)


def get_input():
    """   Get purchase price and quantity from the user   """
    purchase_price = float(input("Enter the purchase price: "))
    quantity = int(input("Enter the quantity: "))
    # Return two values
    return purchase_price, quantity


def calculate_total_sale(price, quantity):
    """   Calculate the total sale   """
    total_sale = price * quantity
    return total_sale


def display_sale(purchase_price, quantity, total_sale):
    """
        Display information about the sale
        {"Purchase Price:":>15 formats string literal
        > align right 15 field width
    """
    print(f'\n{"Purchase Price:":>15} ${purchase_price:,.2f}')
    print(f'{"Quantity:":>15} {quantity}')
    print(25 * "-")
    print(f'{"Total Sale:":>15} ${total_sale:,.2f}')


# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

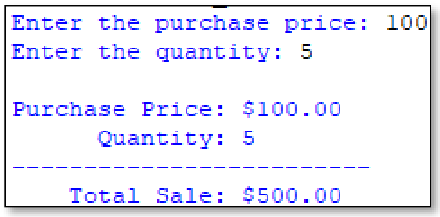
F-strings Formatting of a String Literal

The following line in this tutorial introduces a new concept for fstring formatting, how to format a string literal.

1. Surround the string literal with double quotes: `"Purchase Price:"`
2. Add the F-strings formatting curly braces {} and a colon :
   `{"Purchase Price:":}`
3. Add the format specifiers right align > and field width 15.
   `{"Purchase Price:":>15}`

```
print (f'\n{"Purchase Price:":>15} ${purchase_price:,.2f}')
```

Example run:



```
Enter the purchase price: 100
Enter the quantity: 5

Purchase Price: $100.00
      Quantity: 5
-------------------------
    Total Sale: $500.00
```

Type Hinting

Python is dynamically typed. This means that the type of value that can go into a variable can be changed. We can add typing to Python to be more specific about the types of data that goes into and out of our functions.

Let's take an example of a function which receives two arguments and returns a value indicating their sum:

```
def two_sum(a, b):
    return a + b
```

By looking at this code, one cannot safely and without doubt indicate the type of the arguments for function two_sum. It works both when supplied with int values:

```
print(two_sum(2, 1))  # result: 3
```

and with strings:

```
print(two_sum("a", "b"))  # result: "ab"
```

and with other values, such as lists, tuples et cetera.

Due to this dynamic nature of python types, where many are applicable for a given operation, any type checker would not be able to reasonably assert whether a call for this function should be allowed or not.

To assist our type checker we can now provide type for it in the Function definition indicating the type that we allow.

To indicate that we only want to allow int types we can change our function definition to look like:

```
def two_sum(a: int, b: int):
    return a + b
```

Annotations follow the argument name and are separated by a : character.

Similarly, to indicate only str types are allowed, we'd change our function to specify it:

```
def two_sum(a: str, b: str):
    return a + b
```

Apart from specifying the type of the arguments, one could also indicate the return value of a function call. This is done by adding the -> character followed by the type after the closing parenthesis in the argument list but before the : at the end of the function declaration:

```
def two_sum(a: int, b: int) -> int:
    return a + b
```

We've indicated that the return value when calling two_sum should be of type int. Similarly we can define appropriate values for str, float, list, set and others.

Default Arguments and Keyword Arguments

Python does not have function overloading. When you specify a default value for an argument, that has the same effect.

You can specify a default value for an argument. This makes it optional, and if the caller decides not to use it, then it takes the default value.

You do this by placing an assignment operator and value next to the parameter name, as if it is a regular assignment. When calling the function, you can specify all parameters, or just some of them. In principle the values get passed to the function’s parameters from left to right; if you pass fewer values than there are parameters, as long as default values for the remaining parameters are given, the function call gets executed without runtime errors.

If you define a function with default values for some of the parameters, but not all, it is common to place the ones for which you give a default value to the right of the ones for which you do not.

Here is an example:

```
def multiple_print(string, n=1)
    print(string * n)
    print()

# Similar to function overloading in other languages
# Calling the same function with a different number of arguments
multiple_print("Hello ", 5)
multiple_print("Hello ")
```

```
Hello Hello Hello Hello Hello
Hello
```

Default arguments need to come at the end of the function definition, after the non-default arguments.

A related concept to default arguments is keyword arguments. Say we have the following function definition:

```
def fancy_print( text, color, background, style, justify ):
```

Every time you call this function, you have to remember the correct order of the arguments. Fortunately, Python allows you to name the arguments when calling the function, as shown below:

```
fancy_print( text = "Hi", color = "yellow", background = "black",
style = "bold", justify = "left" )
```

When defining the function, it would be a good idea to give defaults. For instance, if most of the time the caller would want left justification, a white background, etc. Using these values as defaults means the caller does not have to specify every single argument every time they call the function.

Here is an example:

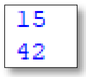
```
def fancy_print( text, color = "black", background = "white",
style = "normal", justify = "left" ):
# function code goes here


fancy_print("Hi", style="bold")
fancy_print("Hi", color="yellow", background="black")
fancy_print("Hi")
```

Functions with Unknown Number of Arguments

Python allows functions with an unknown number of arguments. The argument definition uses the * to indicate that any number of arguments can be sent to the function. The function does have to be written to accommodate an unknown number of arguments.

```
def calculate_total(*arguments):
    total = 0
    for number in arguments:
        total += number
    print(total)
# function calls
calculate_total(5, 4, 3, 2, 1)
calculate_total(35, 4, 3)
```

Example run:

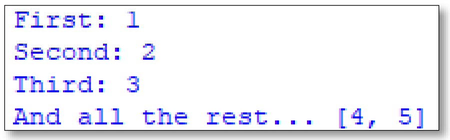


```
15
42
```

You can combine both specific arguments, and multiple arguments.

```
def the_rest(first, second, third, *therest):
    print(f"First: {first}")
    print(f"Second: {second}")
    print(f"Third: {third}")
    print("And all the rest... %s" % list( therest ))


the_rest(1, 2, 3, 4, 5)
```

Example run:



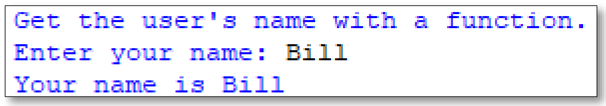
```
First: 1
Second: 2
Third: 3
And all the rest... [4, 5]
```

Returning Strings

Functions can return strings as well as numbers. Here is a simple input example.

```
def get_name():
    """
    Get and return the user's name as a string.
    """
    # Get users name as a string
    name = input("Enter your name: ")
    name = "Your name is " + name
    # Return the string
    return name
name = get_name()
print(name)
```

Example run:



```
Get the user's name with a function.
Enter your name: Bill
Your name is Bill
```

Modules

You have seen how you can reuse code in your program by defining functions once. What if you wanted to reuse functions in other programs that you write? As you might have guessed, the answer is modules.

There are various methods of writing modules, but the simplest way is to create a file with a .py extension that contains functions and variables.

Another method is to write the modules in the native language in which the Python interpreter itself was written.

A module can be imported by another program to make use of its functionality. This is how we can use the Python standard library as well.

We have used import statements in previous programs. These imported modules built into Python. Here are couple of examples of modules we have used before.

The random module is very useful in games.

```
import random
secret_num = random.randint(1, 10)
guess = 0
```

```
while guess != secret_num:
    guess = int(input("Guess the secret number: "))
print("finally got it!")
```

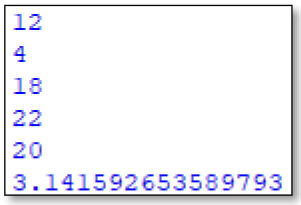
The sys module can be used to exit a program.

```
import sys
ans = input("Quit the program? (Press Enter)")
if ans == ""
    sys.exit()
```

If you would like to use functions from more than one module, you can do so by adding multiple import statements:

```
# Using from to import a specific module,
# you don't have to include the random.randint
# . dot notation
from random import randint
import math
for i in range(5):
    print(randint(1, 25))
print(math.pi)
```

Example run:



```
12
4
18
22
20
3.141592653589793
```

Creating Modules

Creating a module is very simple. You create a Python file, with extension .py, and place functions in it. You can then import this Python file in another Python program (you just use the name of the file without the extension .py; the file should be either in the same folder as the program, or in a standard Python modules location), and access its functions just as you access functions from regular Python modules, i.e., you either import specific functions from the module, or you import the module as a whole, and call its functions by using the <module>.<function>() syntax.

Let's create our first module. Of course, it is Hello World! Let's give this module the filename: **hello.py**

```
# Define a function
def world():
    print("Hello, World!")
```

If we run this program, nothing will happen since we have not told the program to do anything.

Let's create a second file in the same directory called **hello_program.py** so that we can import the module we just created, and then call the function. This file needs to be in the same directory so that Python knows where to find the module since it's not a built-in module.

```
# Import hello module
import hello
# Call function
hello.world()
```

Because we are importing a module, we need to call the function by referencing the module name in dot notation.

Example run:

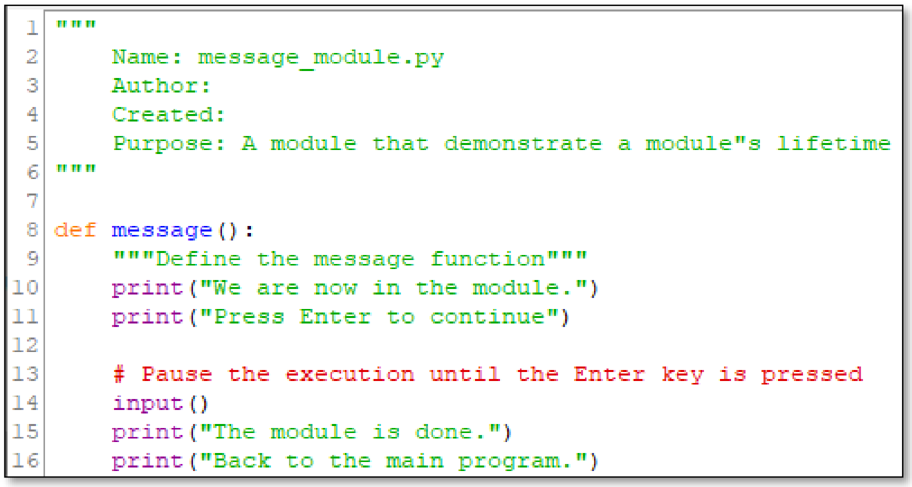
```
Hello, World!
```

Not very impressive, is it? We could have accomplished the same thing with a single program file. Modules will become useful when we start creating larger and more complex programs, or when we want to easily reuse code.

Tutorial 5.5 – A Simple Message Module

Let's create a more complex module that demonstrates the code going from the main program to the module and back again.

1. Create a Python program file named: **message_module.py**
2. Enter the following code.

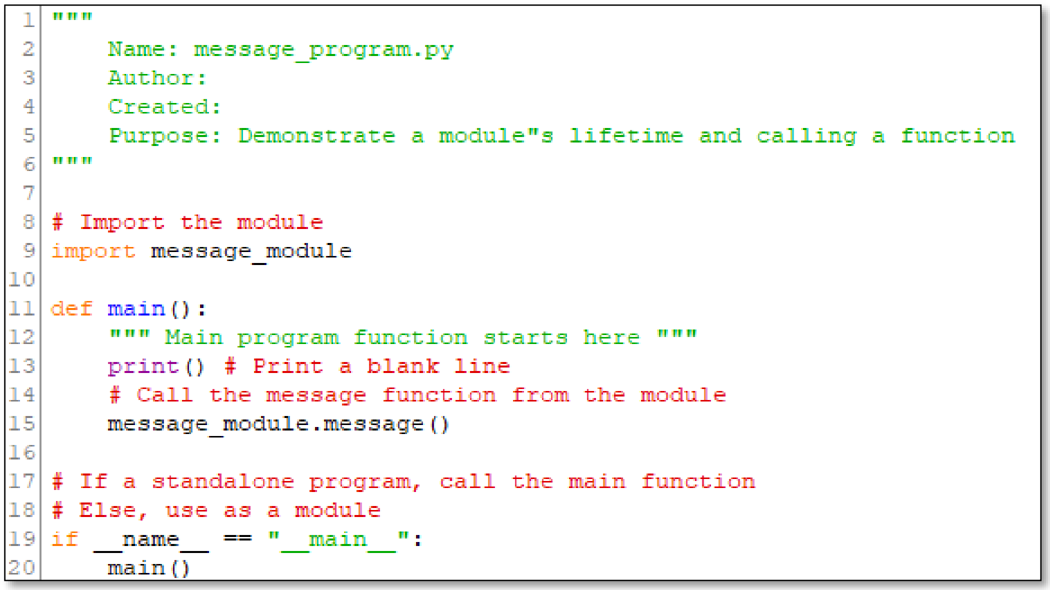


```python
"""
    Name: message_module.py
    Author:
    Created:
    Purpose: A module that demonstrate a module"s lifetime
"""

def message():
    """Define the message function"""
    print("We are now in the module.")
    print("Press Enter to continue")

    # Pause the execution until the Enter key is pressed
    input()
    print("The module is done.")
    print("Back to the main program.")
```

Run the program. Notice that when you run this program, there isn't any output. The function is never called in the program.

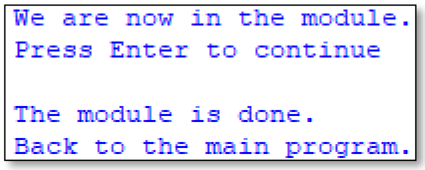
3. Create a Python program file named: **message_program.py**

4. Enter the following code.



```python
"""
    Name: message_program.py
    Author:
    Created:
    Purpose: Demonstrate a module"s lifetime and calling a function
"""

# Import the module
import message_module

def main():
    """ Main program function starts here """
    print() # Print a blank line
    # Call the message function from the module
    message_module.message()

# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

Example run:



```
We are now in the module.
Press Enter to continue

The module is done.
Back to the main program.
```

Is It a Module or a Program?

The answer is: It doesn’t matter. A Python program can be both, it depends on how it is executed.

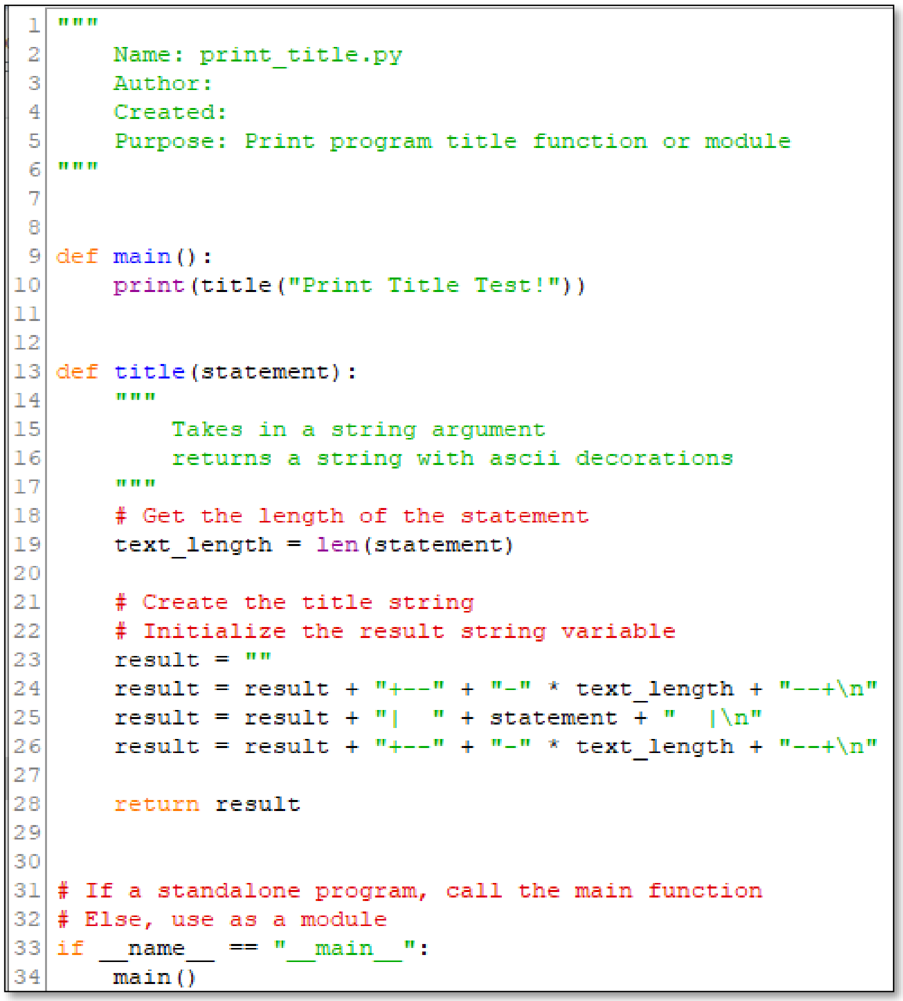
When examining other people’s Python programs, those that contain functions that you might want to import, you often see a construct as shown below:

```
def main():
# code...
if __name__ == "__main__":
    main()
```

The function main() contains the core of the program. It may call other functions. The Python file that contains the code can run as a program, or the functions that it contains can be imported into other programs. The construction shown here ensures that the program only executes main() (which is the core program) if the program is run as a separate program, rather than being loaded as a module. If, instead, the program is loaded as a module into another program, only its functions can be accessed. The code for main() is ignored.

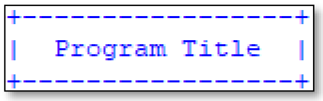
Tutorial 5.6 – Print Title in utils.py

This function can be used as a function in a program, or a module. It is an easy way to print out a descriptive title for a program. You can substitute characters for the border.



```python
"""
    Name: print_title.py
    Author:
    Created:
    Purpose: Print program title function or module
"""


def main():
    print(title("Print Title Test!"))


def title(statement):
    """
        Takes in a string argument
        returns a string with ascii decorations
    """
    # Get the length of the statement
    text_length = len(statement)

    # Create the title string
    # Initialize the result string variable
    result = ""
    result = result + "+--" + "-" * text_length + "--+\n"
    result = result + "|  " + statement + "  |\n"
    result = result + "+--" + "-" * text_length + "--+\n"

    return result


# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

Example run:



```
+-----------------+
|  Program Title  |
+-----------------+
```

Returning Boolean Values

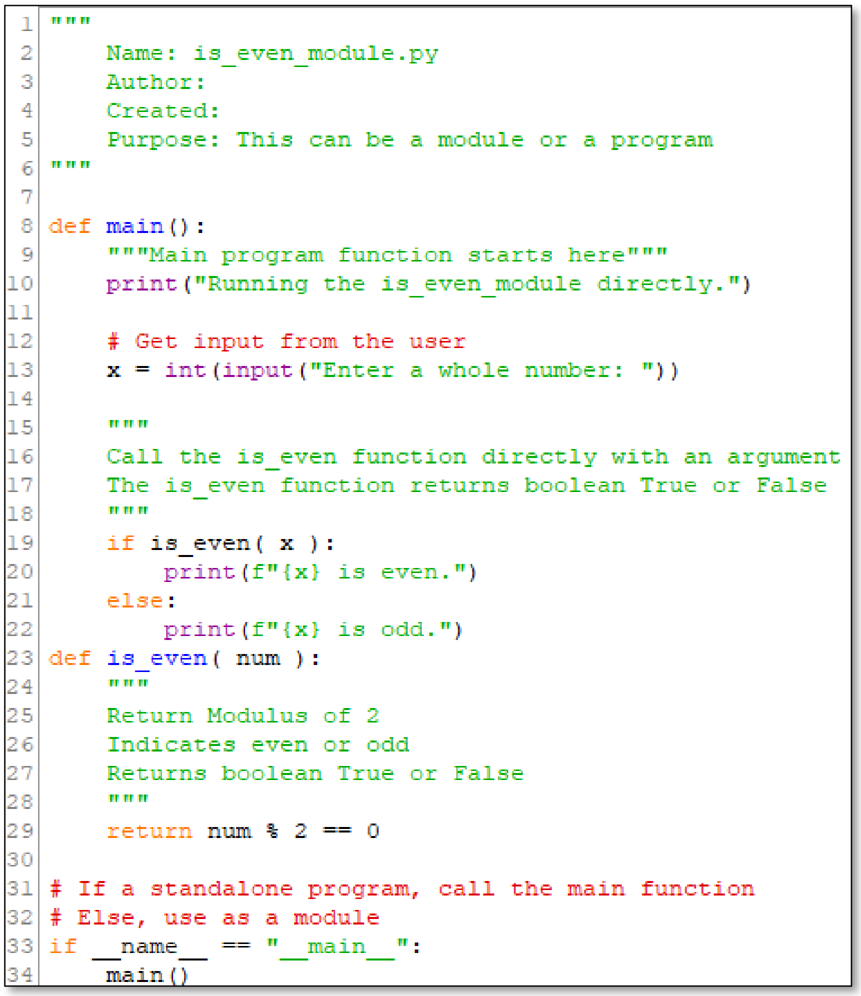
A function can return a Boolean value: True or False. Boolean functions are useful for simplifying complex expressions if decision or repetition structures.

Tutorial 5.7 – is_even Module

Let's create program that returns a Boolean value to a program. This function can also be imported as a module.

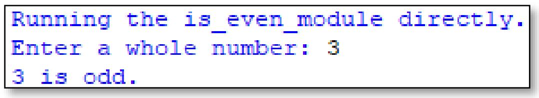
1. Create a Python program file named: **is_even_module.py**

2. Enter the following code.



```python
"""
    Name: is_even_module.py
    Author:
    Created:
    Purpose: This can be a module or a program
"""

def main():
    """Main program function starts here"""
    print("Running the is_even_module directly.")

    # Get input from the user
    x = int(input("Enter a whole number: "))

    """
    Call the is_even function directly with an argument
    The is_even function returns boolean True or False
    """
    if is_even( x ):
        print(f"{x} is even.")
    else:
        print(f"{x} is odd.")
def is_even( num ):
    """
    Return Modulus of 2
    Indicates even or odd
    Returns boolean True or False
    """
    return num % 2 == 0

# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

Example run:

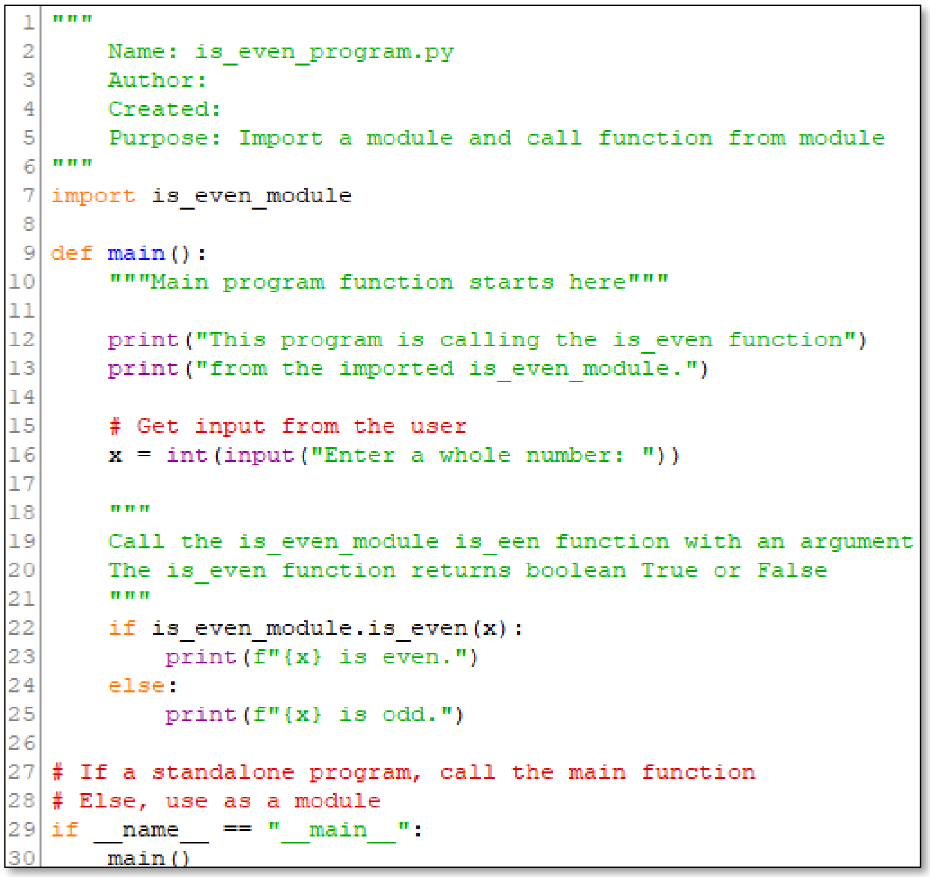


```
Running the is_even_module directly.
Enter a whole number: 3
3 is odd.
```

Notice that the module runs as a program when it is executed directly.

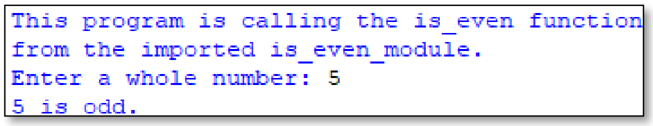
1. Create a Python program file named: **is_even_program.py**

2. Enter the following code.



```python
"""
    Name: is_even_program.py
    Author:
    Created:
    Purpose: Import a module and call function from module
"""
import is_even_module

def main():
    """Main program function starts here"""

    print("This program is calling the is_even function")
    print("from the imported is_even_module.")

    # Get input from the user
    x = int(input("Enter a whole number: "))

    """
    Call the is_even_module is_een function with an argument
    The is_even function returns boolean True or False
    """
    if is_even_module.is_even(x):
        print(f"{x} is even.")
    else:
        print(f"{x} is odd.")

# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

Example run:



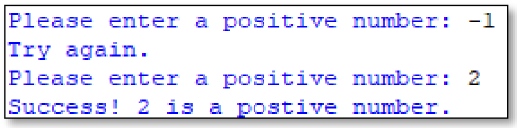
```
This program is calling the is_even function
from the imported is_even_module.
Enter a whole number: 5
5 is odd.
```

Input Validation with Boolean Functions

The following code provides a Boolean return value. Instead of catching program exceptions, this function evaluates whether the input fits the range of the data. This function can be combined with the last input function.

```python
def main():
    number = int(input("Please enter a positive number: "))
    # while the is_invalid function returns true
    # the loop keeps asking for valid input
    while is_invalid(number):
        print("Try again.")
        number = int(input("Please enter a positive number: "))

    print(f"Success! {number} is a positive number. ")

def is_invalid( number ):
    """
        While this function returns true, the data is invalid
    """
    if number < 1:
        status = True
    else:
        status = False

    return status

main()
```

Example run:



```
Please enter a positive number: -1
Try again.
Please enter a positive number: 2
Success! 2 is a postive number.
```

Built-in Python Functions

Python provides several important built-in functions we can use without needing to provide the function definition. The creators of Python wrote a set of functions to solve common problems and included them in Python for us to use.

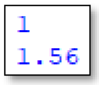
Some functions are called with parameters (“arguments”), which may or may not be mandatory. The parameters are placed between the parentheses that follow the function name. If there are multiple parameters, you place commas between them.

The parameters are the values that the user supplies to the function to work with. For instance, the `int()` function must be called with one parameter, which is the value that the function will try make an integer representation of. The `print()` function may be called with any number of parameters (even zero), which it will display, after which it will go to a new line.

In general, a function cannot change parameters. For instance, look at the following code:

```
x = 1.56
print(int(x))
print(x)
```

Sample run:



```
1
1.56
```

The `int()` function has not changed the actual value of `x`; it only told the `print()` function what the integer value of `x` is. The reason is that, in general, parameters are "passed by value." This means that the function does not get access to the actual parameters, but it gets copies of the values of the parameters.

The max and min functions give us the largest and smallest values in a list, respectively:

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

The `max` function tells us the "largest character" in the string (which turns out to be the letter "w") and the `min` function shows us the smallest character (which turns out to be a space).

Another very common built-in function is the `len` function which tells us how many items are in its argument. If the argument to `len` is a string, it returns the number of characters in the string.

```
>>> len('Hello world')
11
>>>
```

These functions are not limited to looking at strings. They can operate on any set of values.

You should treat the names of built-in functions as reserved words (i.e., avoid using "max" as a variable name).

Type Conversion Functions

Python also provides built-in functions that convert values from one type to another. This can also be called type casting.

The `int` function takes any value and converts it to an integer, if it can, or it complains otherwise:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int() with base 10: 'Hello'
```

`int` converts floating-point values to integers, but it doesn't round off; it chops off the fraction part:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

`float` converts integers and strings to floating-point numbers:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

`str` converts its argument to a string:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

Math Functions

Python has a math module that provides most of the familiar mathematical functions. Before we can use the module, we import it:

```
import math
```

This statement creates a module object named math. If you print the module object, you get some information about it:

```
print(math)
<module 'math' (built-in)>
```

The module object contains the functions and variables defined in the module. To access one of the functions, you specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called dot notation.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)
>>> radians = 0.7
>>> height = math.sin(radians)
```

The first example computes the logarithm base 10 of the signal-to-noise ratio. The math module also provides a function called log that computes logarithms base e.

The second example finds the sine of radians. The name of the variable is a hint that sin and the other trigonometric functions (cos, tan, etc.) take arguments in radians. To convert from degrees to radians, divide by 360 and multiply by 2.

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.7071067811865476
math.pi =
```

The expression `math.pi` gets the variable pi from the math module. The value of this variable approximates π, accurate to about 15 digits.

If you know your trigonometry, you can check the previous result by comparing it to the square root of two divided by two:

```
>>> math.sqrt(2) / 2.0
0.7071067811865476
```

Calculations

Basic Python functions also have limited support for calculations.

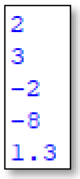
- **abs()** has one numerical parameter (an integer or a float). If the value is positive, it will return the value. If the value is negative, it will return the value multiplied by -1.
- **max()** has two or more numerical parameters, and returns the largest.
- **min()** has two or more numerical parameters, and returns the smallest.

- **pow()** has two numerical parameters, and returns the first to the power of the second. Optionally, it has a third numerical parameter. If that third parameter is supplied, it will return the value modulo that third parameter.
- **round()** has a numerical parameter and rounds it, mathematically, to a whole number. It has an optional second parameter. The second parameter must be an integer, and if it is provided, the function will round the first parameter to the number of decimals specified by the second parameter.

Example program showing each type of calculation.

```
x = -2
y = 3
z = 1.27
print(abs(x))
print(max(x, y, z ))
print(min(x, y, z))
print(pow(x, y))
print(round(z, 1))
```

Example program run:



```
2
3
-2
-8
1.3
```

Random Numbers

Given the same inputs, most computer programs generate the same outputs every time, they are said to be deterministic. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are an obvious example, but there are more.

Making a program truly nondeterministic turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to use algorithms that generate pseudorandom numbers. Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.

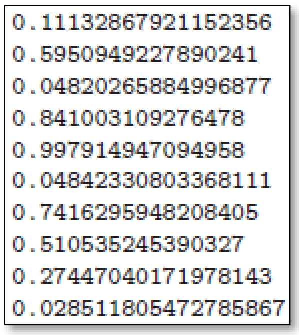
The random module provides functions that generate pseudorandom numbers.

The function random returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time you call random, you get the next number in a long series.

To see a sample, run this loop:

```
import random
for i in range(10):
    x = random.random()
    print(x)
```

This program produces the following list of 10 random numbers between 0.0 and up to but not including 1.0.



```
0.11132867921152356
0.5950949227890241
0.04820265884996877
0.841003109276478
0.997914947094958
0.04842330803368111
0.7416295948208405
0.510535245390327
0.27447040171978143
0.028511805472785867
```

The random function is only one of many functions that handle random numbers. The function **randint** takes the parameters low and high and returns an integer between low and high (including both).

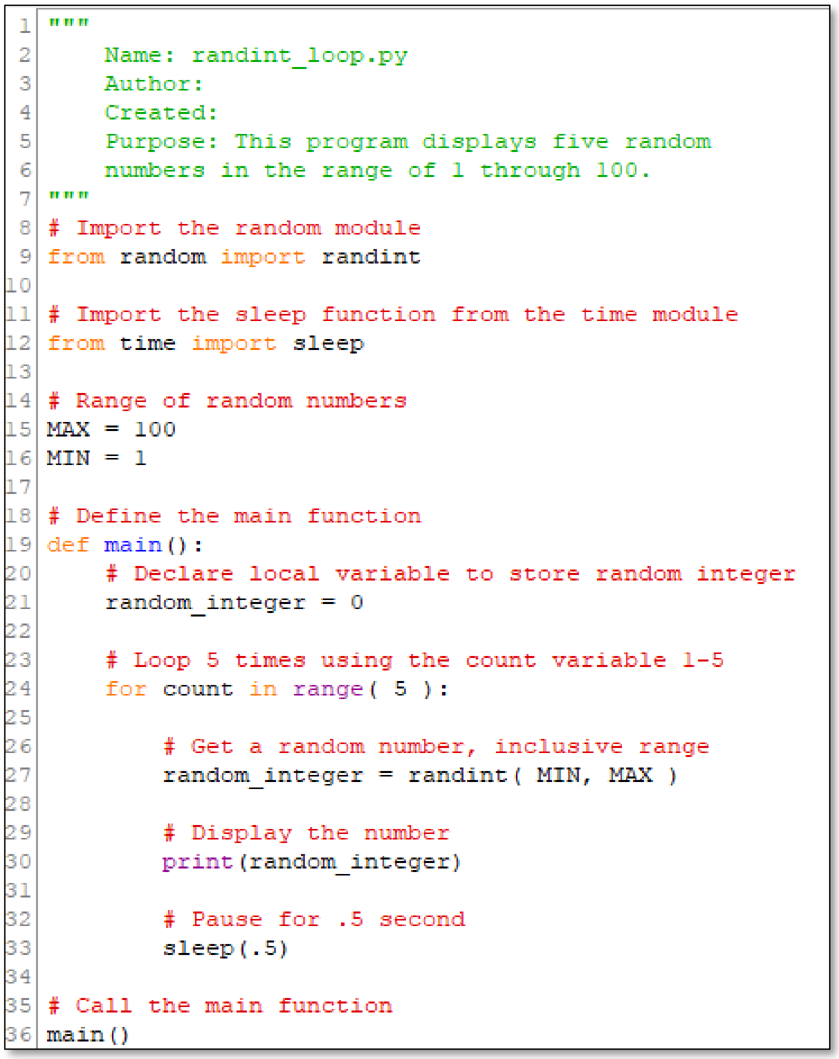
```
From random import randint
for i in range(10):
    x = randint(1, 3)
    print(x)
```

To choose an element from a list at random, you can use choice():

```
import random
# List of three numbers
list = [1, 2, 3]
for i in range(10):
    print(random.choice(list))
```

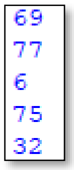
Tutorial 5.8 – Random Numbers in a Loop

The following tutorial demonstrates random integers in a for loop.



```python
"""
    Name: randint_loop.py
    Author:
    Created:
    Purpose: This program displays five random
    numbers in the range of 1 through 100.
"""
# Import the random module
from random import randint

# Import the sleep function from the time module
from time import sleep

# Range of random numbers
MAX = 100
MIN = 1

# Define the main function
def main():
    # Declare local variable to store random integer
    random_integer = 0

    # Loop 5 times using the count variable 1-5
    for count in range( 5 ):

        # Get a random number, inclusive range
        random_integer = randint( MIN, MAX )

        # Display the number
        print(random_integer)

        # Pause for .5 second
        sleep(.5)

# Call the main function
main()
```

Example run:



```
69
77
6
75
32
```

Tutorial 5.9 – Input Function in utils.py Module

NOTE: This bonus tutorial is inspired by a student submission.

We are going to create a **utils.py** module that will contain various functions that we will use in future programs.

The following function gets a numeric input. It uses try except to catch any exceptions. If the input is not the correct data type, the function asks the user again. It passes a prompt parameter to identify what the input function is looking for.

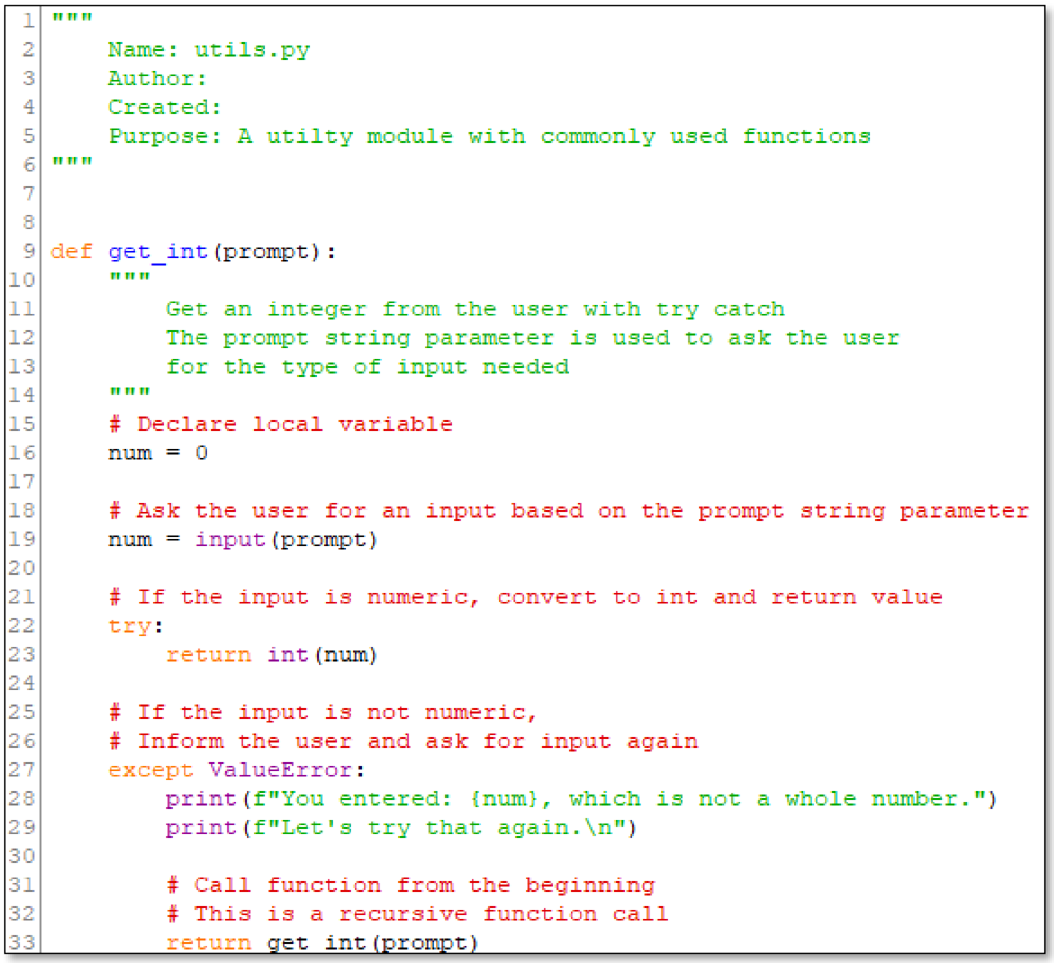
This function can be used in any program that needs int or float numeric input and will catch any exceptions. Change the int to a float in the module.

You don't have to understand how the function works to use it. You can copy and paste it or use it as a module. You just need to know 2 things about how to use the function.

1. The string parameter **what** is the input prompt.
2. The function returns an integer.

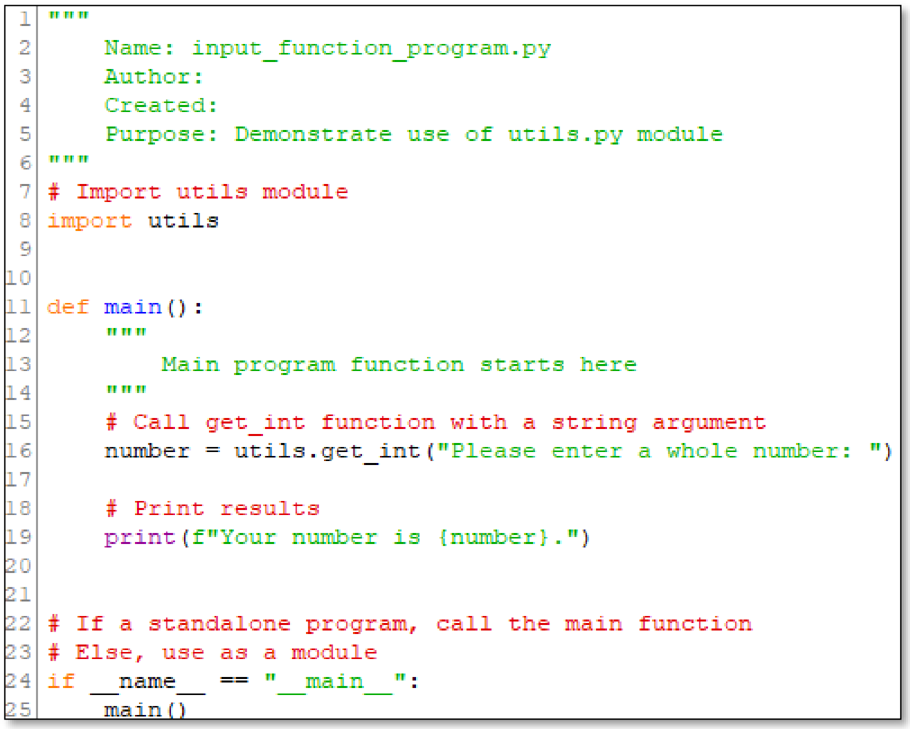
Requirements

1. Open **print_title.py** save it as **utils.py**
2. Add the following code above the title function.



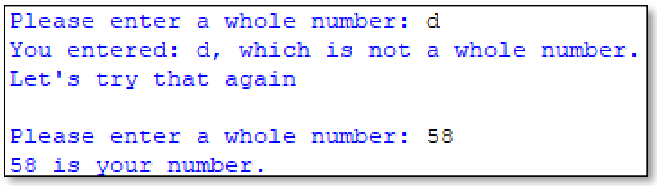
```python
"""
    Name: utils.py
    Author:
    Created:
    Purpose: A utilty module with commonly used functions
"""


def get_int(prompt):
    """
        Get an integer from the user with try catch
        The prompt string parameter is used to ask the user
        for the type of input needed
    """
    # Declare local variable
    num = 0

    # Ask the user for an input based on the prompt string parameter
    num = input(prompt)

    # If the input is numeric, convert to int and return value
    try:
        return int(num)

    # If the input is not numeric,
    # Inform the user and ask for input again
    except ValueError:
        print(f"You entered: {num}, which is not a whole number.")
        print(f"Let's try that again.\n")

        # Call function from the beginning
        # This is a recursive function call
        return get_int(prompt)
```

Example program using **utils.py** as a module.



```python
"""
    Name: input_function_program.py
    Author:
    Created:
    Purpose: Demonstrate use of utils.py module
"""
# Import utils module
import utils


def main():
    """
        Main program function starts here
    """
    # Call get_int function with a string argument
    number = utils.get_int("Please enter a whole number: ")

    # Print results
    print(f"Your number is {number}.")


# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

Example run:



```
Please enter a whole number: d
You entered: d, which is not a whole number.
Let's try that again

Please enter a whole number: 58
58 is your number.
```

Keep the file, **utils.py**, we will continue to use it throughout the class.

Glossary

algorithm A general process for solving a category of problems.

argument A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.

body The sequence of statements inside a function definition.

composition Using an expression as part of a larger expression, or a statement as part of a larger statement.

deterministic Pertaining to a program that does the same thing each time it runs, given the same inputs.

dot notation The syntax for calling a function in another module by specifying the module name followed by a dot (period) and the function name.

flow of execution The order in which statements are executed during a program run.

fruitful function A function that returns a value.

function A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.

function call A statement that executes a function. It consists of the function name followed by an argument list.

function definition A statement that creates a new function, specifying its name, parameters, and the statements it executes.

function object A value created by a function definition. The name of the function is a variable that refers to a function object.

header The first line of a function definition.

import statement A statement that reads a module file and creates a module object.

module object A value created by an import statement that provides access to the data and code defined in a module.

parameter A name used inside a function to refer to the value passed as an argument.

pseudorandom Pertaining to a sequence of numbers that appear to be random but are generated by a deterministic program.

return value The result of a function. If a function call is used as an expression, the return value is the value of the expression.

void function A function that does not return a value.

Assignment Submission

1. Attach the pseudocode.
2. Attach the program files.
3. Attach screenshots showing the successful operation of the program.
4. Submit in Blackboard.